ΕΛΛΗΝΙΚΗ ΔΗΜΟΚΡΑΤΙΑ
Εθνικόν και Καποδιστριακόν Πανεπιστήμιον Αθηνών
ΙΔΡΥΘΕΝ ΤΟ 1837

Course: Machine Learning

ASSIGNMENT: 2

FULL NAME / SERIAL NUMBER: GAVRIELATOS MARIOS / 7115152100023
GIANNAKI EVANGELIA / 7115152100025
SVOLOU STAVROULA / 7115152100038

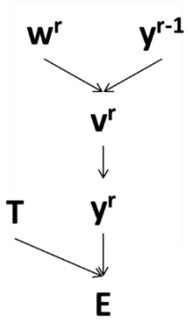
Exercise 1

Jupyter Notebook of Ex.1: *Ex1.ipynb*

Problem A

The cost function used is the following: $E = \frac{1}{2}\sum_{\kappa\mu}(e_{\kappa\mu})^2$ where $e_{\kappa\mu} = T_{\kappa\mu} - y_{\kappa\mu}^{(r)}$ and:

- κ = 1, 2, …, K where K is the number of data
- μ = 1, 2, …, M where M is the number of patters in the training set
- Layer r



Furthermore: $y_{i\mu}^{(r)} = f(v_{i\mu}^r)$ and $v_{i\mu}^r = \sum w_{ij}^r\, y_{j\mu}^{r-1}$

We will use the chain rule for one training example $\frac{\partial E}{\partial w_{ij}^r} = \frac{\partial v_{i\mu}^r}{\partial w_{ij}^r}\frac{\partial y_{i\mu}^{(r)}}{\partial v_{i\mu}^r}\frac{\partial E}{\partial y_{i\mu}^{(r)}}$:

i. $\frac{\partial E}{\partial y_{i\mu}^{(r)}} = T_{ij} - y_{ij}$ the range of which is [0,1] because we use softmax as the output layer,

ii. $\frac{\partial y_{i\mu}^{(r)}}{\partial v_{i\mu}^r} = f'(v_{i\mu}^r)$ the range of which depends of the activation function,

iii. $\frac{\partial v_{i\mu}^r}{\partial w_{ij}^r} = y_{j\mu}^{r-1}$ the range of which depends from the range of the activation function of the previous layer.

1) **ReLU**: $f(x) = \max(0, x)$ and range [0, +∞].

The gradient, $\frac{\partial y_{i\mu}^{(r)}}{\partial v_{i\mu}^r}$: $\begin{cases} 1\ for\ \ v \geq 0 \\ 0\ f0r\ v < 0 \end{cases} \Rightarrow \begin{matrix} \frac{\partial E}{\partial w_{ij}^r} = y_{j\mu}^{r-1}\cdot 1\cdot(T_{ij} - y_{ij}) \\ \frac{\partial E}{\partial w_{ij}^r} = y_{j\mu}^{r-1}\cdot 0\cdot(T_{ij} - y_{ij}) \end{matrix} \Rightarrow \begin{matrix} \frac{\partial E}{\partial w_{ij}^r} = y_{j\mu}^{r-1}\cdot 1\cdot(T_{ij} - y_{ij})\ \ for\ v \geq 0 \\ \frac{\partial E}{\partial w_{ij}^r} = 0\ for\ v < 0 \end{matrix}$

Thus the range of the gradient $\frac{\partial E}{\partial w_{ij}^r}$ is [0, +∞].

2) **Hyperbolic Tangent (tanh)**: $f(x) = \frac{e^x - e^{-x}}{e^x - e^{-x}}$ and range [-1, +1].

Derivative of tanh: $\frac{\partial f(x)}{\partial x} = 1 - f^2(x)$ with range [0, +1].

When $\frac{\partial y_{i\mu}^{(r)}}{\partial v_{i\mu}^r} = 1$: $\frac{\partial E}{\partial w_{ij}^r} = y_{j\mu}^{r-1} \cdot 1 \cdot (T_{ij} - y_{ij})$ with range [-1, +1] because the range of $y_{j\mu}^{r-1}$is [-1, +1].

When $\frac{\partial y_{i\mu}^{(r)}}{\partial v_{i\mu}^r} = 0$: $\frac{\partial E}{\partial w_{ij}^r} = 0$

Thus, the range of the gradient $\frac{\partial E}{\partial w_{ij}^r}$ is [-1, +1].

3) **Sigmoid**: $f(x) = \frac{1}{1 + e^{-x}}$ and range [0, 1].

Derivative of sigmoid: $\frac{\partial f(x)}{\partial x} = f(x)[1 - f(x)]$ with range [0, 0.25].

When $\frac{\partial y_{i\mu}^{(r)}}{\partial v_{i\mu}^r} = \frac{1}{1 + e^{-v_{i\mu}^r}}\left(1 - \frac{1}{1 + e^{-v_{i\mu}^r}}\right)$: $\frac{\partial E}{\partial w_{ij}^r} = y_{j\mu}^{r-1} \cdot \frac{1}{1 + e^{-v_{i\mu}^r}}\left(1 - \frac{1}{1 + e^{-v_{i\mu}^r}}\right) \cdot (T_{ij} - y_{ij})$

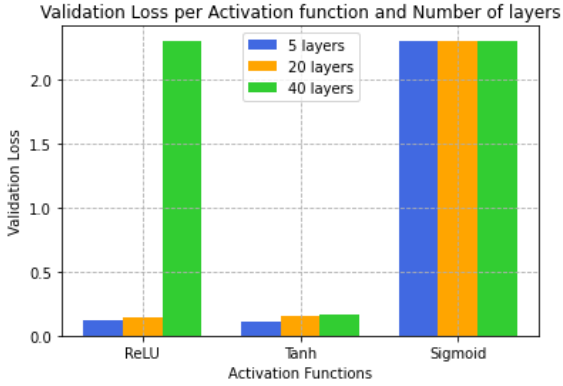
When $\frac{\partial y_{i\mu}^{(r)}}{\partial v_{i\mu}^r} = 0$: $\frac{\partial E}{\partial w_{ij}^r} = 0$

Thus, the range of the gradient $\frac{\partial E}{\partial w_{ij}^r}$ is [0, 0.25].

Problem B

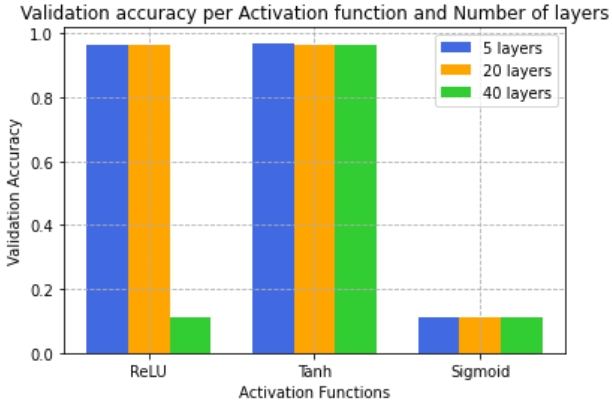
The first table includes the test **loss** from 5, 20 and 40 layers using 3 different activation functions (ReLU, Tanh and Sigmoid). The network was trained for 40 epochs.

#Hidden Layers	Test Loss		
	ReLU	Tanh	Sigmoid
5	0.1211	0.1042	2.3012
20	0.1454	0.1546	2.3014
40	2.301	0.1652	2.3012



The first table includes the test **accuracy** from 5, 20 and 40 layers using 3 different activation functions (ReLU, Tanh and Sigmoid). The network was trained for 40 epochs.

#Hidden Layers	Test Accuracy		
	ReLU	Tanh	Sigmoid
5	0.9651	0.9698	0.1135
20	0.9617	0.9632	0.1135
40	0.1135	0.9615	0.1135

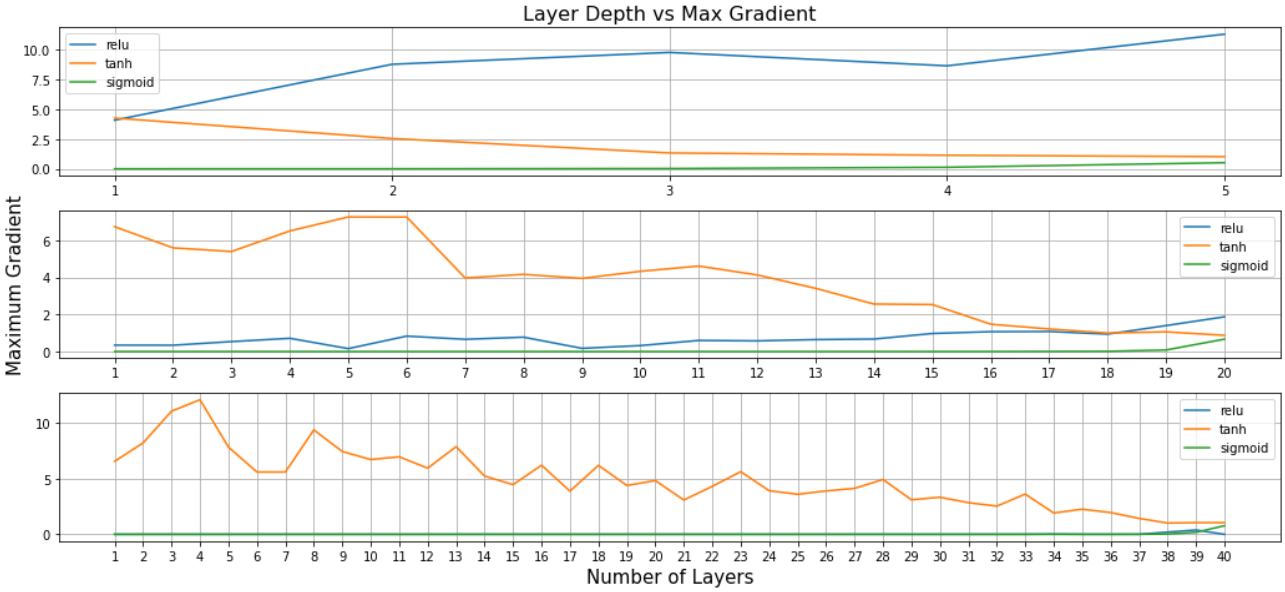


Observing the above plots and tables, we are able to conclude that the highest accuracy and lowest loss are achieved using 5 Hidden layers. Using Tanh activation function, we observe that there is not a significance decrease in accuracy as we add hidden layers to the model, however we observe a significant decrease using the ReLU activation function and 40 layers. The Sigmoid function shows the lowest accuracy.

These results show that as we increase the number of layers, and thus the complexity of the network, the network is not able to generalize well. Furthermore, regarding the Sigmoid function, it is important to note that the gradient, at either tail of 0 or 1, is almost zero. Very small values of the gradient lead to considerable slowdown of the convergence of the gradient descent algorithms.

Problem C

The graph below shows the maximum values of the gradient per layer, for 3 different networks (5, 20 and 40 layers), given a specific mini-batch using the 3 different activation functions. The network was trained for 3 epochs.



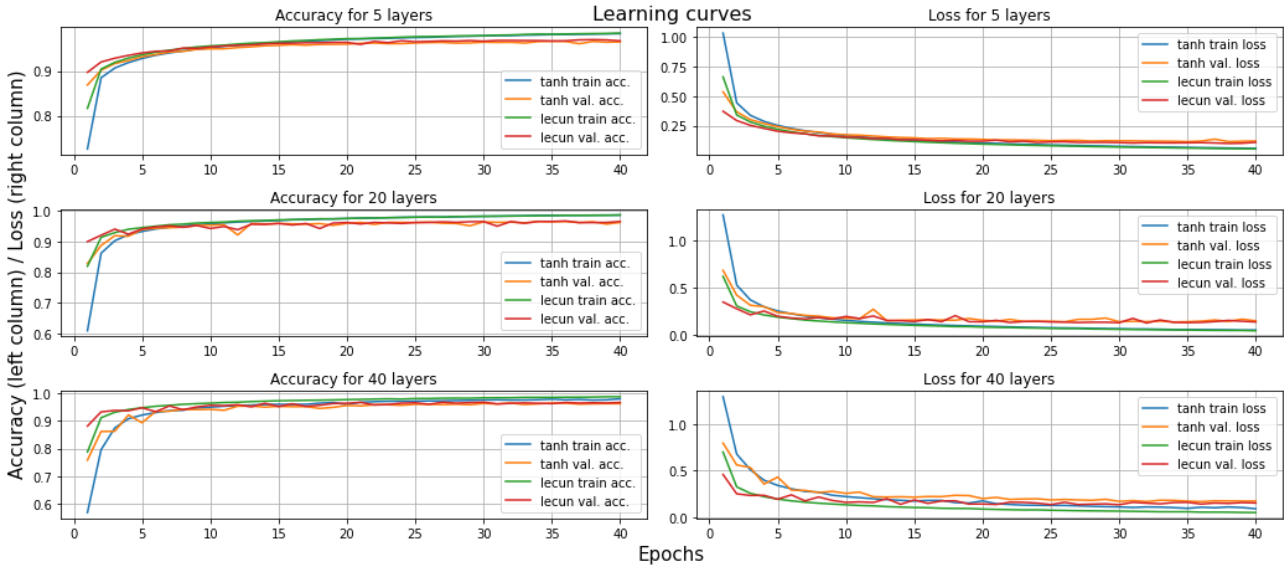
Observing the above plots, we infer that the gradients of the **Sigmoid** function (green lines) are close to zero in the 3 different networks. This happens, as we have already discussed, because the sigmoid neurons get saturated on the boundaries of the function and thus the local gradients at these regions are almost zero. The large positive values are squashed near 1 and large negative values are squashed near 0. Hence, effectively making the local gradient to near 0. This leads to “kill” the gradients.

Moreover, we observe that the gradients of the **Tanh** function (orange line) are not close to zero in the 3 different networks and remain relatively constant. The same applies for the gradients of the **ReLU** function (blue line) for the 5- and 20-layer network. On the other hand, the gradients of the ReLU function on the 40-layer network converge to zero. This does not happen for the same reason as in the Sigmoid case but because of small gradient values.

These results are in accordance with the results of Problem B.

Problem D

The following figure shows the learning curves for the train and the validation set for the Tanh function and the LeCun function trained for 40 epochs. The left plots show the accuracy and the right plots the loss of the models per epoch. The learning curves of the two functions seem similar.



The cost function used in this case is the categorical cross entropy cost function:

$E = -\sum_{k\mu} T_{k\mu} \log\left(y_{k\mu}^{(r)}\right)$.

Furthermore: $y_{i\mu}^{(r)} = f(v_{i\mu}^{r})$ and $v_{i\mu}^{r} = \sum w_{ij}^{r}\, y_{j\mu}^{r-1}$

We will use again the chain rule for one training example $\frac{\partial E}{\partial w_{ij}^{r}} = \frac{\partial v_{i\mu}^{r}}{\partial w_{ij}^{r}} \frac{\partial y_{i\mu}^{(r)}}{\partial v_{i\mu}^{r}} \frac{\partial E}{\partial y_{i\mu}^{(r)}}$:

i. $\frac{\partial E}{\partial y_{i\mu}^{(r)}} = -\frac{T_{ij}}{y_{ij}}$ the range of which is $[-\infty, 0]$ because we use softmax as the output layer and thus T_{ij} and y_{ij} have a range of [0, 1],

ii. $\frac{\partial y_{i\mu}^{(r)}}{\partial v_{i\mu}^{r}} = f'(v_{i\mu}^{r})$ the range of which depends of the activation function,

iii. $\frac{\partial v_{i\mu}^{r}}{\partial w_{ij}^{r}} = y_{j\mu}^{r-1}$ the range of which depends from the range of the activation function of the previous layer.

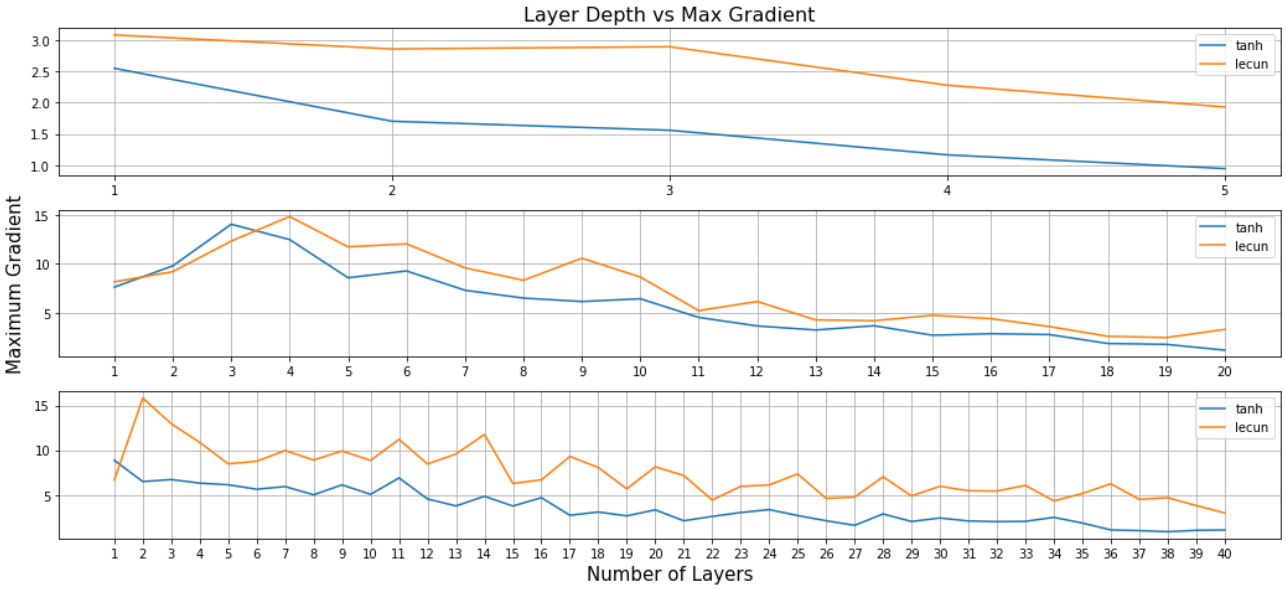
The **LeCun** activation function: $f(x) = 1.7159 \cdot tanh\left(\frac{2}{3}x\right) + 0.01 \cdot x$ with range [-∞, +∞].

The derivative of the LeCun: $\frac{\partial f(x)}{\partial x} = 1.1439 \cdot \left(1 - tanh^2\left(\frac{2}{3}x\right)\right) + 0.01$ with range [0.01, 1.1539]. This occurs because $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{1 - e^{-2x}}{1 + e^{-2x}}$, $\lim_{x\to\pm\infty}(1 - e^{-2x}) = 1$ and $\lim_{x\to\pm\infty}(1 + e^{-2x}) = 1$. Thus, $\lim_{x\to\pm\infty} \tanh(x) = 1$. From these we can conclude that $\frac{\partial y_{i\mu}^{(r)}}{\partial v_{i\mu}^{r}}$ has a range [0.01, 1.1539].

Because LeCun activation function has a range of [-∞, +∞], the term $\frac{\partial v_{i\mu}^{r}}{\partial w_{ij}^{r}}$ also has a range of [-∞, +∞].

Therefore, the range of the gradient $\frac{\partial E}{\partial w_{ij}^{r}}$ is [-∞, +∞].

The graph below shows the maximum values of the gradient per layer, for 3 different networks (5, 20 and 40 layers), given a specific mini-batch using the 2 different activation functions. The network was trained for 3 epochs. We can observe that in all networks, the gradients of the Tanh function are smaller than the LeCun function.

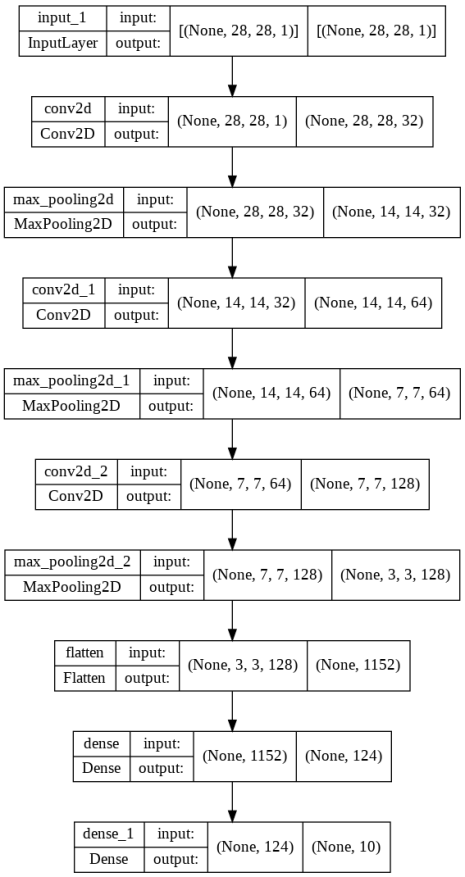


Exercise 2

Problem A

Jupyter Notebook of Problems A and B: *Ex2_mnist.ipynb*

The convolutional neural network that we designed and trained on the MNIST dataset consists of three convolutional layers and a fully connected neural network. Each convolutional layer includes the Convolution, the ReLU activation function and Max-pooling. After the convolutional layers, there is a layer that flattens the image and a fully connected neural network consisting of a hidden layer with a ReLU activation function and an output layer with a softmax activation function. The model was compiled to use the Adam optimizer and the Cross-entropy loss. A validation set was used to avoid overfitting; therefore, we trained the model until it reached an accuracy greater than 99.3% on the validation set.



Problem B

The MNIST-simple-generator f_{ij} is a fully connected neural network that contains the input layer (only one input neuron equal to 1), two hidden layers and one output layer. The output is always a 28x28 grayscale image. We train this model using each time the specific sample from the test set, i, that we want to use as target. The model was compiled using the Adam optimizer and the mean squared error loss. Until this point, we have a trained model that takes as input the number 1 and outputs an image similar to sample i. Then, we created a composite model $C \circ f$ and we train it with target, j, equal to one of the 10 classes. We repeated this procedure for all 10 classes for each sample and for 5 different samples (0, 1, 2, 4, 7). Figures 1 2 3 4 5 show the classification of the adversarial examples created. We are able to see that with the added noise the model misclassifies the digits every time. However, a human is still able to classify the digit correctly.

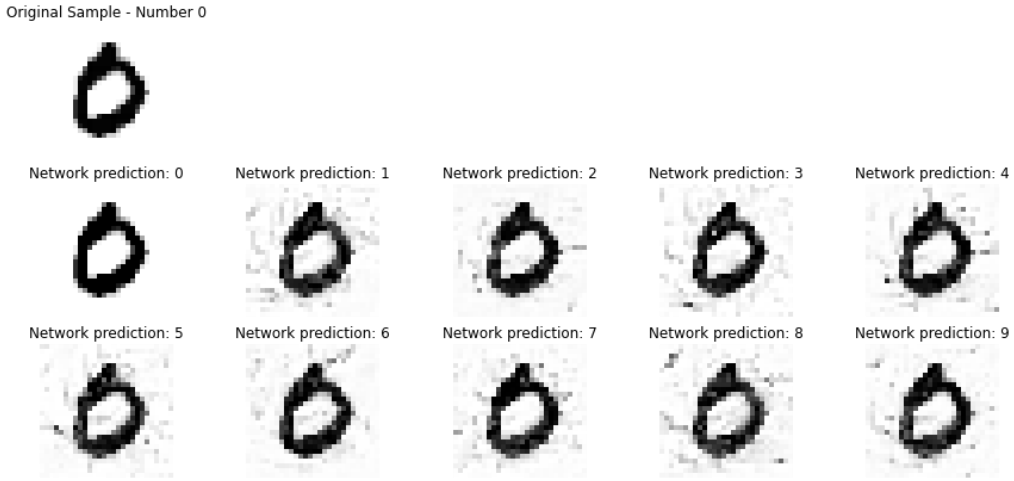


Figure 1: *Adversarial examples of digit 0.*

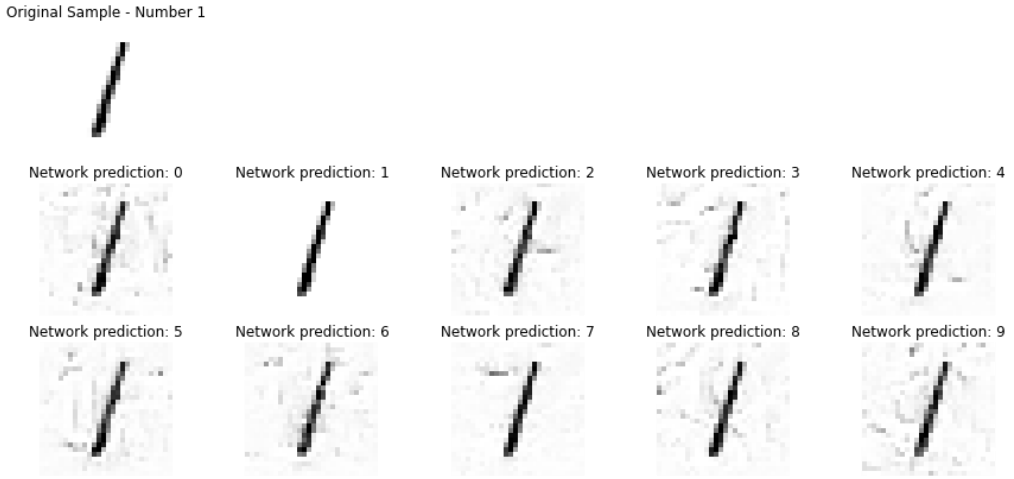


Figure 2*: Adversarial examples of digit 1.*

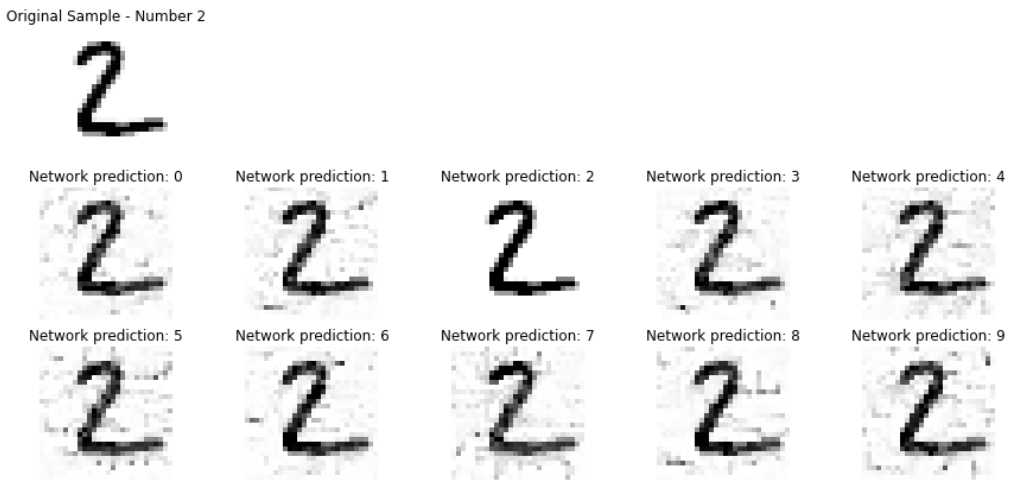


Figure 3*: Adversarial examples of digit 2.*

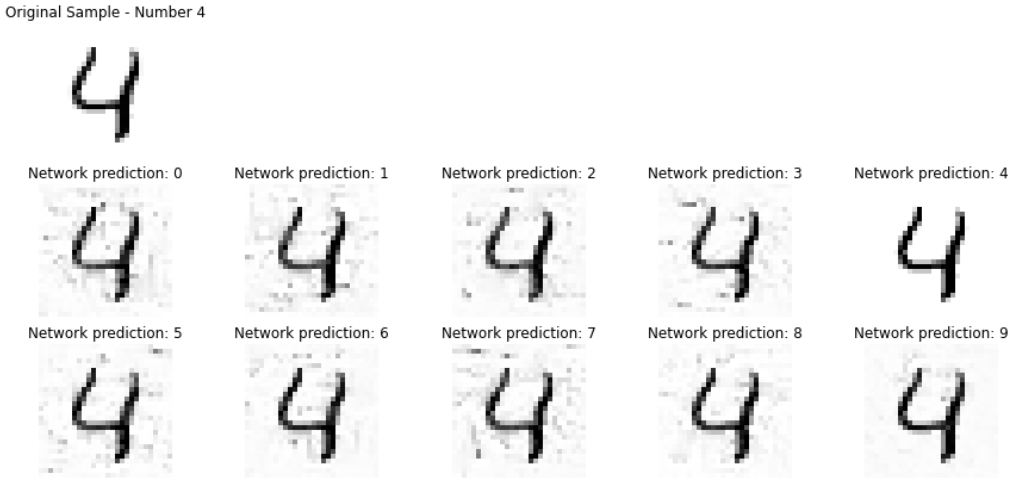


***Figure 4**: Adversarial examples of digit 4.*

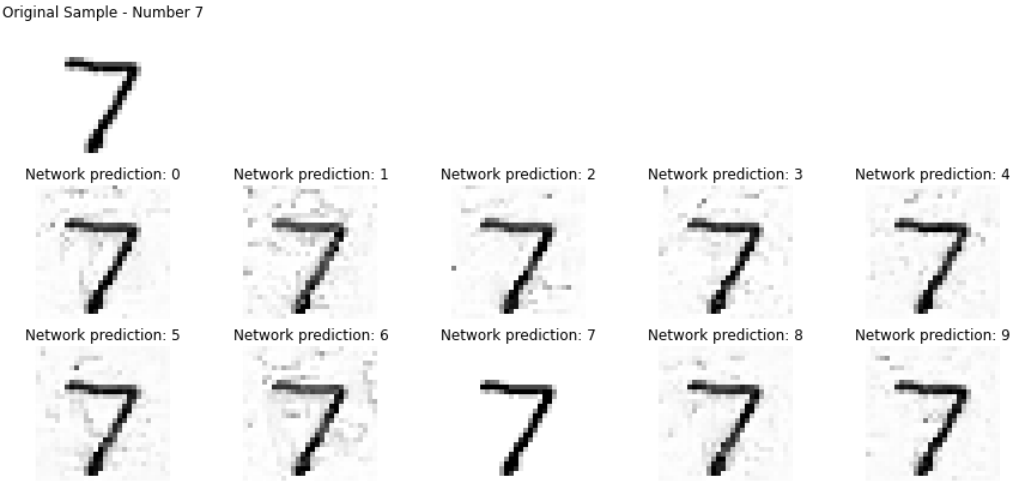
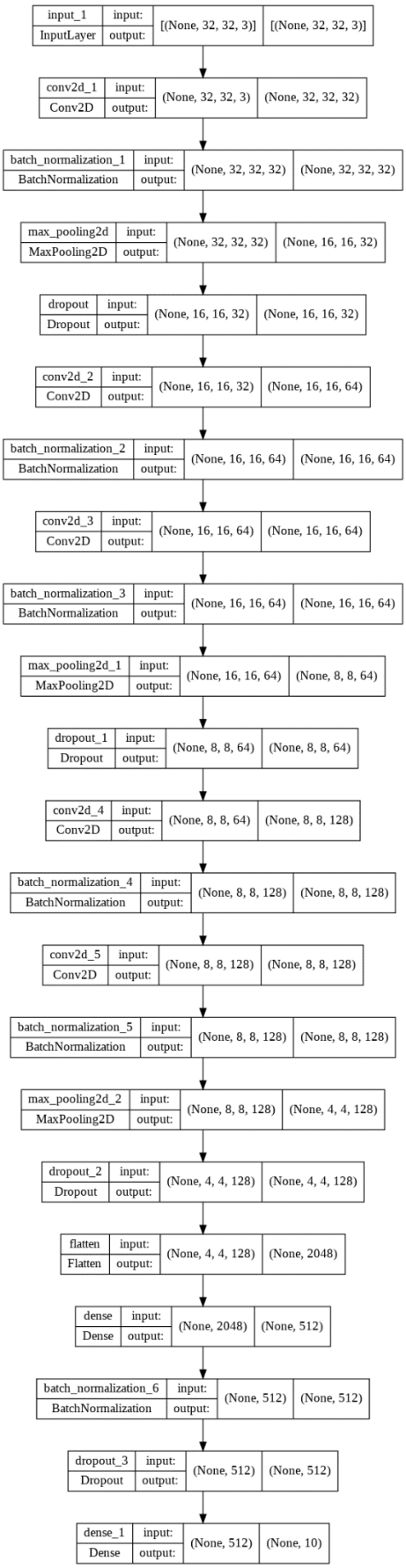


***Figure 5**: Adversarial examples of digit 7.*

Problem C

Jupyter Notebook of Problem C: *Ex2_cifar.ipynb*

In the case of CIFAR10 dataset, we followed the same procedure with some alterations. The CIFAR10 dataset is much more complex, therefore we designed a more complex convolutional neural network in order to achieve a satisfying accuracy on the validation set. This network also has 3 convolutional layers, however each of these layers consist of the convolution, the ReLU activation function, batch normalization (in order to reduce overfitting), another set of convolution-ReLU-batch normalization, max-pooling and dropout. The dropout was added to avoid overfitting. After the convolutional layers, there is a layer that flattens the image and a fully connected neural network consisting of a hidden layer with a ReLU activation function, batch normalization, dropout (50%) and an output layer with a softmax activation function. The model uses the Adam optimizer and the Cross-entropy loss function. In order to achieve an even better accuracy, we performed data augmentation to create both training and validation data by flipping the images horizontally, shifting the width of the images and shifting their height. We trained the model for 100 epochs and the classifier achieved an accuracy of 88.49.


input_1 | InputLayer | input: | output: | [(None, 32, 32, 3)] | [(None, 32, 32, 3)]
conv2d_1 | Conv2D | input: | output: | (None, 32, 32, 3) | (None, 32, 32, 32)
batch_normalization_1 | BatchNormalization | input: | output: | (None, 32, 32, 32) | (None, 32, 32, 32)
max_pooling2d | MaxPooling2D | input: | output: | (None, 32, 32, 32) | (None, 16, 16, 32)
dropout | Dropout | input: | output: | (None, 16, 16, 32) | (None, 16, 16, 32)
conv2d_2 | Conv2D | input: | output: | (None, 16, 16, 32) | (None, 16, 16, 64)
batch_normalization_2 | BatchNormalization | input: | output: | (None, 16, 16, 64) | (None, 16, 16, 64)
conv2d_3 | Conv2D | input: | output: | (None, 16, 16, 64) | (None, 16, 16, 64)
batch_normalization_3 | BatchNormalization | input: | output: | (None, 16, 16, 64) | (None, 16, 16, 64)
max_pooling2d_1 | MaxPooling2D | input: | output: | (None, 16, 16, 64) | (None, 8, 8, 64)
dropout_1 | Dropout | input: | output: | (None, 8, 8, 64) | (None, 8, 8, 64)
conv2d_4 | Conv2D | input: | output: | (None, 8, 8, 64) | (None, 8, 8, 128)
batch_normalization_4 | BatchNormalization | input: | output: | (None, 8, 8, 128) | (None, 8, 8, 128)
conv2d_5 | Conv2D | input: | output: | (None, 8, 8, 128) | (None, 8, 8, 128)
batch_normalization_5 | BatchNormalization | input: | output: | (None, 8, 8, 128) | (None, 8, 8, 128)
max_pooling2d_2 | MaxPooling2D | input: | output: | (None, 8, 8, 128) | (None, 4, 4, 128)
dropout_2 | Dropout | input: | output: | (None, 4, 4, 128) | (None, 4, 4, 128)
flatten | Flatten | input: | output: | (None, 4, 4, 128) | (None, 2048)
dense | Dense | input: | output: | (None, 2048) | (None, 512)
batch_normalization_6 | BatchNormalization | input: | output: | (None, 512) | (None, 512)
dropout_3 | Dropout | input: | output: | (None, 512) | (None, 512)
dense_1 | Dense | input: | output: | (None, 512) | (None, 10)

For the adversarial examples we used the same adversarial model as in the case of MNIST and we defined the composite model C ◦ f. Figure 6 depicts the classification of the CIFAR10 adversarial examples. As in Problem B, the model misclassifies the image every time we add the appropriate noise for each class. In this case, we are not able to see the noise added to the images, in contrast to the handwritten digits. This happens because of the more complex nature of RGB images of the CIFAR10 dataset.

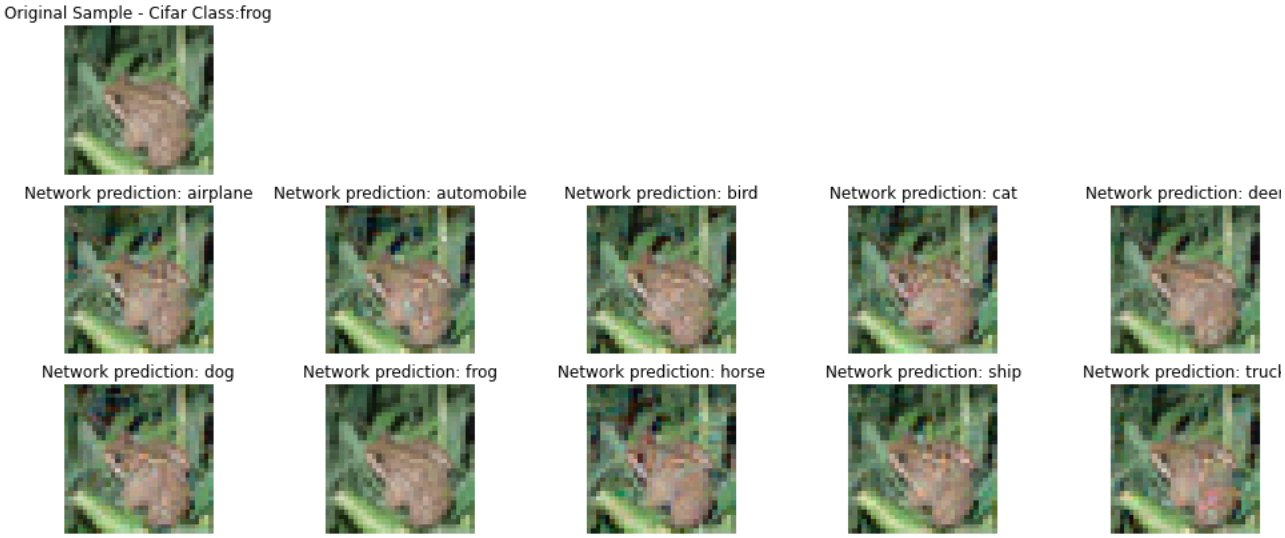


***Figure 6**: Adversarial examples of the class "frog".*

Exercise 3

Jupyter Notebook of Exercise 3: *Ex3.ipynb*

Model: *mask_detection_model.h5*

As it has been proved scientifically, a way to avoid the spread of respiratory diseases is by wearing face masks. For the time being, where we have already confronted two years of the coronavirus pandemic (COVID-19), it became clear the importance of applying that precautionary measure meticulously. In this framework, plenty of times it is necessary to decide quickly and accurately from a picture, the existence or not of a face mask.

Below we are going to develop a classifier, which will achieve face mask detection, by taking into consideration a various set of images, where are depicted people with or without wearing a face mask. To be more specific, to train and validate our classifier we utilized α dataset derived from Kaggle. The dataset (https://www.kaggle.com/andrewmvd/face-mask-detection) consists of 853 images with 4072 faces, that are separated into 3 classes. The classes are the following:

1. With mask: 3232 images
2. Without mask: 123 images
3. Mask worn improperly: 717 images

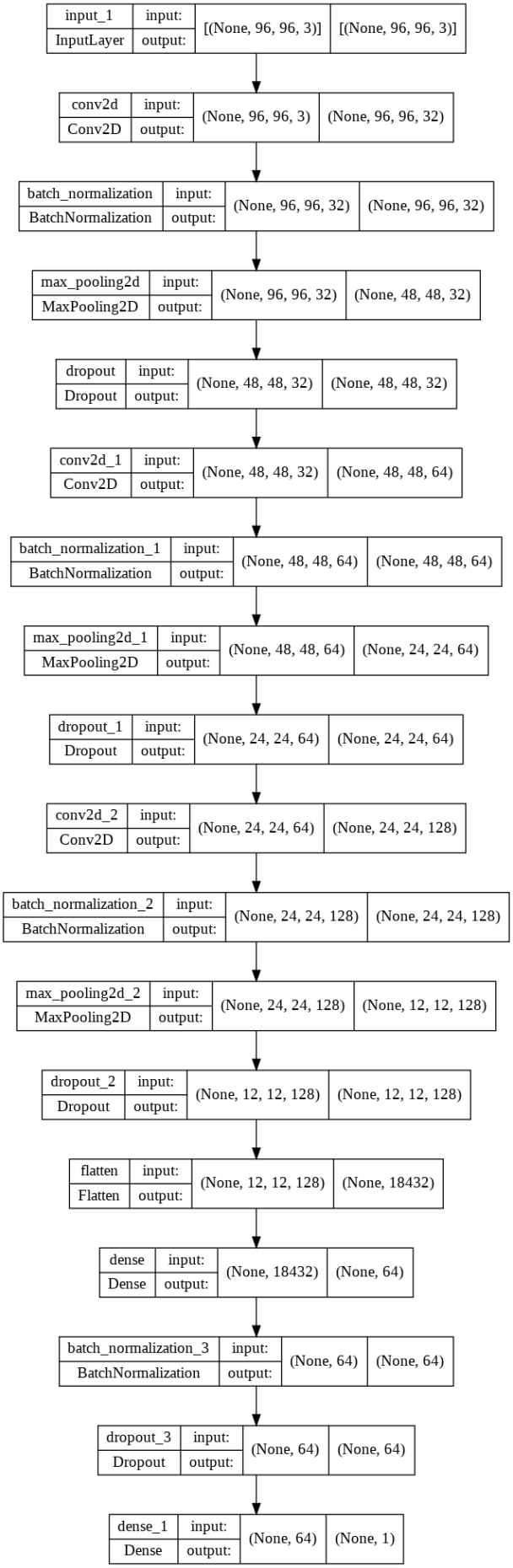
The third class was incorporated into the second class. 10% of the pictures from each class were used as the validation set.

We train our classifier to separate the images into two classes, individuals that wear a mask and those that do not wear, regardless of the way that is applied (properly or incorrectly). As we can see in code, the model includes 3 Convolutional layers, where each of them consists of the 4 following “steps”:

1. Conv2D
2. Batch Normalization
3. Max Pooling
4. Dropout

We add a fully connected Dense layer of 64 units followed by BatchNormalization and Dropout. For the aforementioned layers we used the LeakyReLU function as an activation function. Finally, we add a Dense layer of 1 unit with the sigmoid activation function. As a loss function we used the binary_crossentropy function and the Adam optimizer with learning rete 0.0005.

We trained the model until it reached validation accuracy greater than 97% and validation loss less than 1%. In the following figure we demonstrate the structure of our model:


input_1 | InputLayer | input: | output: | [(None, 96, 96, 3)] | [(None, 96, 96, 3)]
conv2d | Conv2D | input: | output: | (None, 96, 96, 3) | (None, 96, 96, 32)
batch_normalization | BatchNormalization | input: | output: | (None, 96, 96, 32) | (None, 96, 96, 32)
max_pooling2d | MaxPooling2D | input: | output: | (None, 96, 96, 32) | (None, 48, 48, 32)
dropout | Dropout | input: | output: | (None, 48, 48, 32) | (None, 48, 48, 32)
conv2d_1 | Conv2D | input: | output: | (None, 48, 48, 32) | (None, 48, 48, 64)
batch_normalization_1 | BatchNormalization | input: | output: | (None, 48, 48, 64) | (None, 48, 48, 64)
max_pooling2d_1 | MaxPooling2D | input: | output: | (None, 48, 48, 64) | (None, 24, 24, 64)
dropout_1 | Dropout | input: | output: | (None, 24, 24, 64) | (None, 24, 24, 64)
conv2d_2 | Conv2D | input: | output: | (None, 24, 24, 64) | (None, 24, 24, 128)
batch_normalization_2 | BatchNormalization | input: | output: | (None, 24, 24, 128) | (None, 24, 24, 128)
max_pooling2d_2 | MaxPooling2D | input: | output: | (None, 24, 24, 128) | (None, 12, 12, 128)
dropout_2 | Dropout | input: | output: | (None, 12, 12, 128) | (None, 12, 12, 128)
flatten | Flatten | input: | output: | (None, 12, 12, 128) | (None, 18432)
dense | Dense | input: | output: | (None, 18432) | (None, 64)
batch_normalization_3 | BatchNormalization | input: | output: | (None, 64) | (None, 64)
dropout_3 | Dropout | input: | output: | (None, 64) | (None, 64)
dense_1 | Dense | input: | output: | (None, 64) | (None, 1)

In order to run the Jupyter notebook of this exercise you will need a kaggle.json file to download the Kaggle dataset.